LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-408738

CLOMP: Accurately Characterizing OpenMP Application Overheads

G. Bronevetsky, J. Gyllenhaal, B. R. de Supinski

November 14, 2008

International Journal of Parallel Programming

Disclaimer

CLOMP: Accurately Characterizing OpenMP Application Overheads

Greg Bronevetsky, John Gyllenhaal, and Bronis R. de Supinski
Computation Directorate
Lawrence Livermore National Laboratory
Livermore, CA 94551, USA
greg@bronevetsky.com, gyllen@llnl.gov, bronis@llnl.gov

Abstract. Despite its ease of use, OpenMP has failed to gain widespread use on large scale systems, largely due to its failure to deliver sufficient performance. Our experience indicates that the cost of initiating OpenMP regions is simply too high for the desired OpenMP usage scenario of many applications. In this paper, we introduce CLOMP, a new benchmark to characterize this aspect of OpenMP implementations accurately. CLOMP complements the existing EPCC benchmark suite to provide simple, easy to understand measurements of OpenMP overheads in the context of application usage scenarios. Our results for several OpenMP implementations demonstrate that CLOMP identifies the amount of work required to compensate for the overheads observed with EPCC. We also show that CLOMP also captures limitations for OpenMP parallelization on SMT and NUMA systems. Finally, CLOMPI, our MPI extension of CLOMP, demonstrates which aspects of OpenMP interact poorly with MPI when MPI helper threads cannot run on the NIC.

1 Introduction

OpenMP [11] is a simple method to incorporate shared memory parallelism into scientific applications. While OpenMP has grown in popularity, it has failed to achieve widespread usage in those applications despite the use of shared memory nodes as the building blocks of large scale resources on which they run. Many factors contribute to this apparent contradiction, most of which reflect the failure of OpenMP-based applications to realize the performance potential of the underlying architecture. First, the applications run on more than one node of these large scale resources and, thus, the applications use MPI [10]. While distributed shared memory OpenMP implemetantions [9] are an option, they fail to provide the same level of performance.

Application programmers still might have adopted a hybrid OpenMP/MPI style, using OpenMP for on-node parallelization. However, the performance achieved discourages that also. OpenMP programs often have higher Amdahl's fractions than with MPI for on-node parallelization. Optimization of OpenMP usage has proven difficult due to a lack of a standard OpenMP profiling interface and, more so, to a myriad of confusing and often conflicting environment settings that govern OpenMP performance. In addition, the lack of on-node parallelization within MPI implementations has often implied higher network bandwidths

[0] This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (UCRL-ABS-XXXXXX).

with multiple MPI tasks on a node. Perhaps the most important factor has been a mismatch between the amount of work in typical OpenMP regions of scientific applications and the overhead of starting those regions.

Multi-core systems will impact many factors that have restricted adoption of OpenMP. Future networking hardware will not support the messaging rates required to achieve reasonable performance with an MPI task per core. Also, greater benefit from on-node parallelization within MPI implementations will provide similar (or better) aggregate network bandwidth to hybrid OpenMP/MPI applications compared to using an MPI task per core. Further, shared caches will provide memory bandwidth benefits to threaded applications.

Since we expect OpenMP to gain popularity with future large scale systems, we must understand the impact of OpenMP overheads on realistic application regions. Accurately characterizing them will help motivate chip designers to provide hardware support to reduce them if necessary. In this paper, we present CLOMP, a new benchmark that complements the EPCC suite [13] to capture the impact of OpenMP overheads (the CLOMP benchmark has no relationship to Intel's Cluster OpenMP). CLOMP is a simple benchmark that models realistic application code structure, and thus the associated limits on compiler optimization. We use CLOMP to model several application usage scenarios on a range of current shared memory systems. Our results demonstrate that OpenMP overheads limit performance substantially for large scale multiphysics applications and that NUMA effects can dramatically lower their performance even when they can compensate for those overheads. We also find that an existing simultaneous multithreading (SMT) implementation provides little benefit to realistic OpenMP scenarios. We then present CLOMPI, our extension of CLOMP to capture the impact on OpenMP overheads of a hybrid OpenMP/MPI programming model. Our results show that the impact is often insubstantial, particularly when MPI helper threads run on the NIC. The impact can be much more significant without that capability although SMT support on the cores can reduce it.

2 Characteristics of Scientific Applications

CLOMP provides a single easy-to-use benchmark that captures the shared memory parallelization characteristics of a wide range of scientific applications. We focused on applications in use at Lawrence Livermore National Laboratory (LLNL), which are representative of large scale applications. We categorize LLNL applications as multiphysics applications or as science applications that focus on a particular physics domain. We need a simple easy-to-use benchmark that accurately characterizes the performance that a system and its OpenMP implementation will deliver to the full range of these applications.

Multiphysics applications [4, 5, 14, 16] generally have large, complex code bases with multiple code regions that contribute significantly to their total run time. These routines occur in disparate application code sections as well as third party libraries, such as linear solvers [1, 6]. While the latter may include large loops that are relatively amenable to OpenMP parallelization, the application code often has many relatively small but parallelizable loops with dependencies

between the loops that inhibit loop fusion to increase the loop sizes. Further, the loops frequently occur in disparate function calls related to different physics packages, making consolidation even more difficult. Many multiphysics applications use unstructured grids, which imply significant pointer chasing to retrieve the actual data. Code restructuring to overcome these challenges is difficult: not only are these applications typically very large (a million lines of code or more) but the exact routines and the order in which they are executed depends on the input. However, the individual loops have no internal dependencies and would appear to be good candidates for OpenMP parallelization.

Science applications typically have fewer lines of code and less diverse execution profiles. While many still use high performance numerical libraries such as ScaLAPACK [2], a single routine often contains the primary computational kernel. Loop sizes available for OpenMP parallelization vary widely, from dense large matrix operations to very short loops. LLNL science applications include first principles molecular dynamics codes [8], traditional molecular dynamics codes [7, 12, 15] and ParaDiS, a dislocation dynamics application [3].

The loop sizes available for OpenMP parallelization depend on the application and the input problem. Currently, many HPC applications either use weak scaling or increase the problem resolution, both of which imply the loop sizes do not vary substantially as the total number of processors increases. However, we anticipate systems with millions of processor cores in the near future, which will make strong scaling attractive. Further, the amount of memory per core will decrease substantially. Both of these factors will lead to smaller OpenMP loops. Thus, while we need an OpenMP benchmark that characterizes the range of applications, capturing the impact of decreasing loop sizes is especially important.

Both multiphysics and science applications run at large scale, using MPI to communicate between nodes and either MPI or shared memory for on-node parallelism; the later leads to hybrid OpenMP/MPI applications. Many hybrid applications use a simple phased approach in which they alternate between OpenMP and MPI regions. Since MPI plays an important role in evaluating the performance of these hybrid applications, we also require a benchmark that evaluates the interaction of compute-intensive threads with features of the MPI runtime such as message-passing communication and any extra threads that implement MPI communication. Multiphysics applications use a wide variety of communication patterns, with stencil patterns being particularly common. A full evaluation of possible patterns is beyond the scope of a simple benchmark so we limit our consideration to a ring communication pattern, which is a straightforward one dimensional stencil with periodic boundaries.

3 The CLOMP Benchmark Implementation

CLOMP is structured like a multiphysics application. Its state mimics an unstructured mesh with a set of partitions, each divided into a linked list of zones, as Figure 1 shows. The linked lists limit optimizations but we allocate the zones contiguously so CLOMP can benefit from prefetching. The amount of memory allocated per zone can be adjusted to model different pressures on the memory system; however, computation is limited to the first 32 bytes of each zone. We

kept the per-zone working set constant because many applications only touch a subset of a zone's data on each pass, including our target applications. Although the actual size varies from application to application, our constant size simplifies exploration of interactions between the CPU and the memory subsystem.

CLOMP repeatedly executes the loop shown in Figure 2. `calc_deposit()` represents a synchronization point, such as an MPI call or a computation that depends on the state of all partitions. The subsequent loop contains `numPartitions` independent iterations. Each iteration traverses a partition's linked list of zones, depositing a fraction of a substance into each zone. We tune the amount of computation per zone by repeating the inner loop `flopScale` times.

CLOMP models several possible loop parallelization methods, outlined in Figure 3. The first applies a combined `parallel for` construct to the outer loop, using either a `static` or a `dynamic` schedule. We call these configurations `for-static` and `for-dynamic`. The second method, called `manual`, represents parallelization that the programmer can perform manually to reduce the Amdahl's fraction. We enclose all instances of CLOMP's outer loop in a `parallel` construct and partition each work loop among threads explicitly. To ensure correct execution, we follow the work loop by a `barrier` and enclose the `calc_deposit` in a `single` construct. The last configuration, called `best-case` represents the optimistic scenario in which all OpenMP synchronization is instantaneous. It is identical to the `manual` version, except that the `barrier` and `single` are removed. Although this configuration would not produce correct answers, it provides an upper bound for the performance improvements possible for the other configurations.

While similar to the schedule benchmark in EPCC that measures the overhead of the loop construct with different schedule kinds, CLOMP emulates application scenarios through several parameters in order to characterize the impact of that overhead. The `numPartitions` parameter determines the number of independent pieces of work in each outer loop while the `numZonesPerPart` and the `flopScale` parameters determine the amount of work in each partition. While our results in Section 4.2 fix `numPartitions` to 64, we can vary it as appropriate

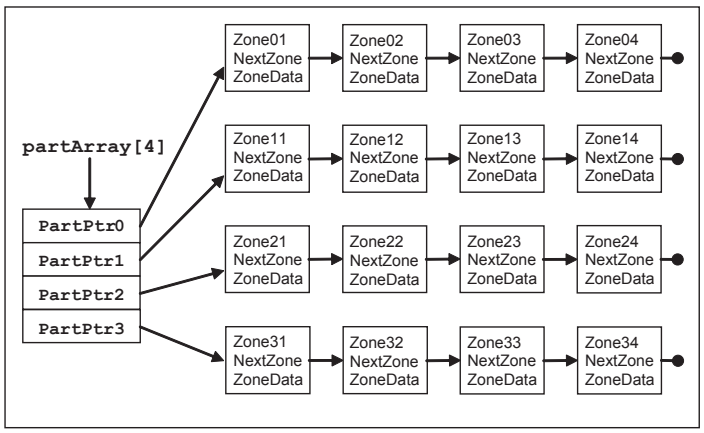

Fig. 1. CLOMP data structures

```
deposit = calc_deposit();
for(part = 0; part < numPartitions; part++) {
  for(zone = partArray[part]->firstZone; zone != NULL; zone = zone->nextZone) {
    for(scale_count = 0; scale_count < flopScale; scale_count++) {
      deposit = remaining_deposit * deposit_ratio;
      zone->value += deposit;
      remaining_deposit -= deposit; } } }
```

Fig. 2. CLOMP source code

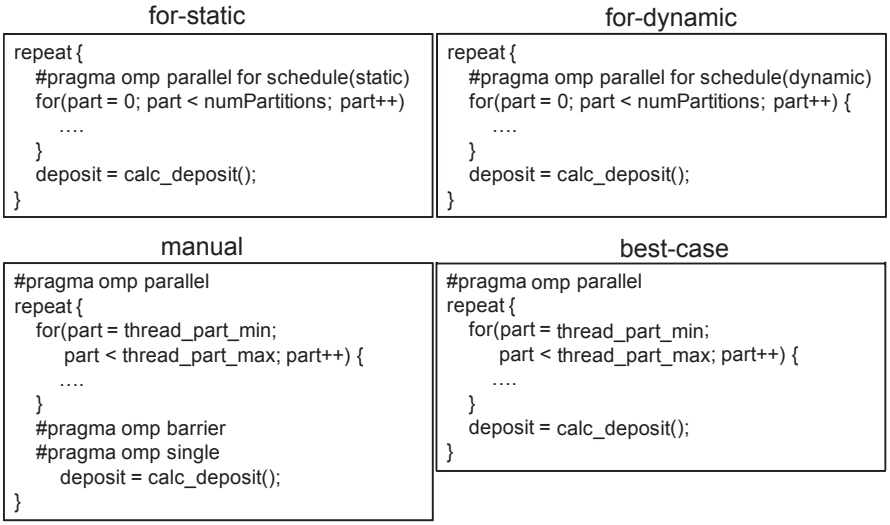

Fig. 3. Variants of CLOMP

for the application being modeled. The EPCC test fixes the corresponding factor at 128 per thread and requires source code modification to vary it; which prevents direct investigation of speed ups for a loop with a fixed total amount of work. The EPCC test also fixes the amount of work per iteration to approximately 100 cycles; our results show that this parameter directly impacts the speed up achieved. CLOMP could mimic the EPCC schedule benchmark through proper parameter settings but those would not correspond to any application scenarios likely to benefit from OpenMP parallelization.

Our results in Section 4.2 demonstrate that we must measure the impact of memory issues as well as the schedule overheads alone to capture the effectiveness of an OpenMP implementation for many realistic application loops. We control CLOMP's memory footprint through the `zoneSize` parameter that specifies the amount of memory allocated per zone. In addition, the `allocThreads` parameter determines whether each thread allocates its own partitions or if the master thread allocates all of the partitions. As is well known, the earlier strategy works better on NUMA systems that employ a first touch policy to place pages.

CLOMPI provides two variants of CLOMP that evaluate the impact of MPI on OpenMP-parallelized applications. The first, CLOMPI-No_Comm. is identical to CLOMP, except that it calls `MPI_Init()` at the beginning of the run and `MPI_Finalize()` at the end. These calls ensure that the MPI runtime, including any additional threads, runs concurrently with CLOMP's OpenMP tests. CLOMPI-SendRecv, the second variant, includes actual MPI communication in the form of a single `MPI_Sendrecv()` operation in the `calc_deposit()` routine. This communication moves an `MPI_DOUBLE` value one step along a virtual ring that includes all the application nodes.

4 Experimental Results

In this section, we demonstrate that CLOMP provides the context of application OpenMP usage for results obtained with the EPCC microbenchmarks [13] through results on three different shared memory nodes. The LLNL Atlas system has dual core, quad socket (8-way) 2.4GHz Opteron, 16GB main memory nodes. Each core has 64KB L1 instruction and data caches and a 1MB L2 cache; each dual core chip has a direct connection to 4GB of local memory with HyperTransport connections to the memories of the other chips. The LLNL Thunder system has 4-way 1.4GHz Itanium2, 4GB main memory nodes. Each single core chip has 16KB instruction and data caches, a 256KB L2 cache and a 4MB L3 cache. All four processors on a node share access to main memory through four memory hubs. Our experiments on Thunder and Atlas use the Intel compiler version 9.1, including its OpenMP run time. The LLNL uP system has dual core, quad socket (8-way) 1.9GHz Power5, 32 GB main memory nodes. Each core has private 64KB instruction and 32KB data caches while a 1.9MB L2 cache and a 36MB L3 cache are shared between the two cores on each chip. Each dual core chip has a direct connection to 8GB of local memory with connections through the other chips to their memories. Our experiments on uP use the IBM xlc compiler version 7.0, including its OpenMP run time.

All experiments on all platforms use the `-O3` optimization level. We used thread affinity to ensure each thread used a different core but the threads were not bound, meaning that the operating system could move them. We relied on the kernel's memory affinity algorithm to keep memory close to the threads that allocated it but the exact details of the algorithms used are unknown.

4.1 OpenMP Overheads Measured with EPCC

We measured the overheads of OpenMP constructs on our target platforms with the EPCC microbenchmark suite. Figure 4 presents the results of the synchronization microbenchmark and Figure 5 show the scheduling microbenchmark. All figures list OpenMP constructs on the x-axis and their average overhead from ten runs in processor cycles on the y-axis. The synchronization benchmark data is plotted on a linear y-axis and the scheduling data uses a logarithmic axis.

The synchronization microbenchmark data shows several interesting effects. First, while the overheads of synchronization constructs with Intel OpenMP vary little with the number of threads, they rise dramatically as the number of threads increases with IBM OpenMP. However, despite its poor scaling, IBM OpenMP

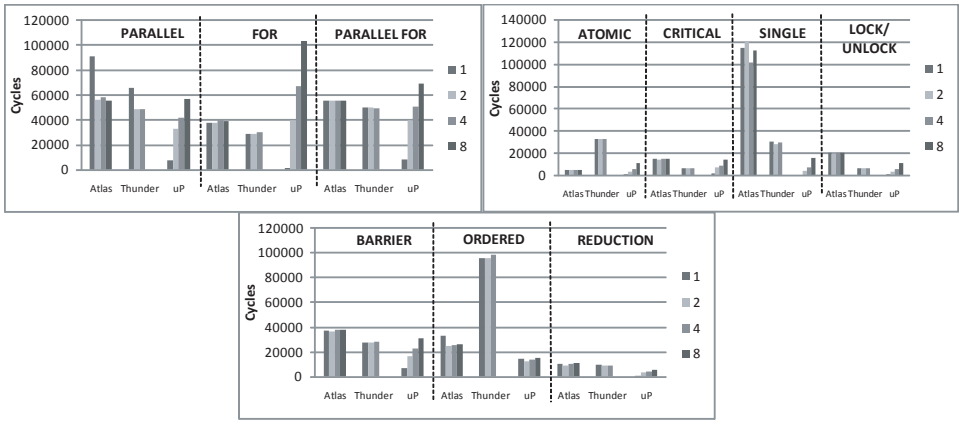

Fig. 4. EPCC Synchronization Results

is less expensive for most OpenMP constructs The exceptions are the `atomic` and `critical` and parallel loop constructs, which have higher overhead with IBM OpenMP on larger thread counts. Overall, most synchronization overheads are on the order of tens of thousands of cycles. In particular, a `barrier` costs between 27,000 and 38,000 cycles with Intel OpenMP and from 7,000 to 31,000 with IBM OpenMP. The overhead of a loop construct is 28,000-40,000 cycles with Intel OpenMP and ranges from 1,400 to 100,000 cycles with IBM OpenMP. The overhead of a combined parallel loop construct is typically a little larger than the maximum overhead of the separate constructs.

The overhead of different schedule kinds varies between our platforms also, as shown in Figures 5 (the y-axis is logarithmic). The overhead of the loop construct changes little as the number of threads increases with our two Intel OpenMP platforms for a fixed schedule kind and associated chunk size. Further, static scheduling overhead is similar for all chunk sizes. In contrast, dynamic scheduling overhead drops off exponentially with increasing chunk size while guided scheduling overhead falls linearly. The reduced overheads reflect that the dynamic and guided mechanisms impose a cost every time they are invoked. Since larger chunks imply fewer invocations of the chunk assignment mechanism, they impose a smaller overhead. This drop-off is less pronounced for guided scheduling because it uses smaller chunks at the end of the allocation process, while dynamic scheduling uses similar chunk sizes throughout. Nonetheless, dynamic and guided scheduling overheads are consistently higher than static scheduling overhead on the Intel OpenMP platforms, ranging from twice as high with a chunk size of 128 to a factor of ten higher on Thunder and 50 on Atlas with a chunk size of one. On Thunder, guided scheduling overhead with a chunk size of 32 is 1.8x lower than the static scheduling overhead; the reason for this is unclear. The overheads of different schedule kinds with IBM OpenMP rise superlinearly with the number of threads. However, IBM OpenMP overheads

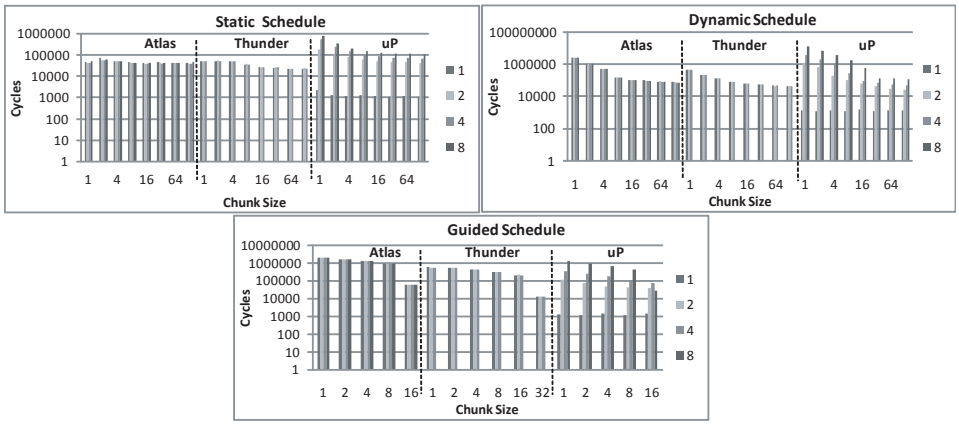

Fig. 5. EPCC Scheduling Results

exhibit the same patterns with respect to chunk size patterns as seen with Intel OpenMP except that static scheduling shows even steeper overhead drops than dynamic and guided scheduling with increasing chunk size. In addition, static scheduling overhead is not much lower than the other schedule kinds with the same chunk size and is sometimes larger.

The EPCC results capture the relative cost of different schedule kinds on our platforms. When compared to Intel OpenMP, IBM OpenMP with dynamic and guided scheduling is always cheaper with one thread and is usually cheaper with two. In all other cases, IBM OpenMP is more expensive as its poor scalability overtakes its good sequential performance. The results demonstrate that users should use static scheduling with Intel OpenMP unless their loop bodies have very significant load imbalances while, with IBM OpenMP, the more flexible schedule kinds are more likely to prove worthwhile. However, these low level EPCC results do not include sufficient information to determine if an application can compensate for the overheads. While it helps to convert the overheads to cycles from the microseconds that the test suite reports, we still need measures that capture the effect of these overheads for realistic application scenarios.

4.2 Capturing the Impact of OpenMP Overheads with CLOMP

We model application scenarios through CLOMP parameter settings. All results presented here set `numPartitions` to 64 and `flopScale` to 1. CLOMP's default parameters, including `numZonesPerPart` equal to 100, model the relatively small loop sizes of many multiphysics application. The defaults use the minimum zone size of 32 bytes, which provides the most opportunity for prefetching and limits its memory system pressure, and have the master thread allocate all memory similarly to the usual default in most applications.

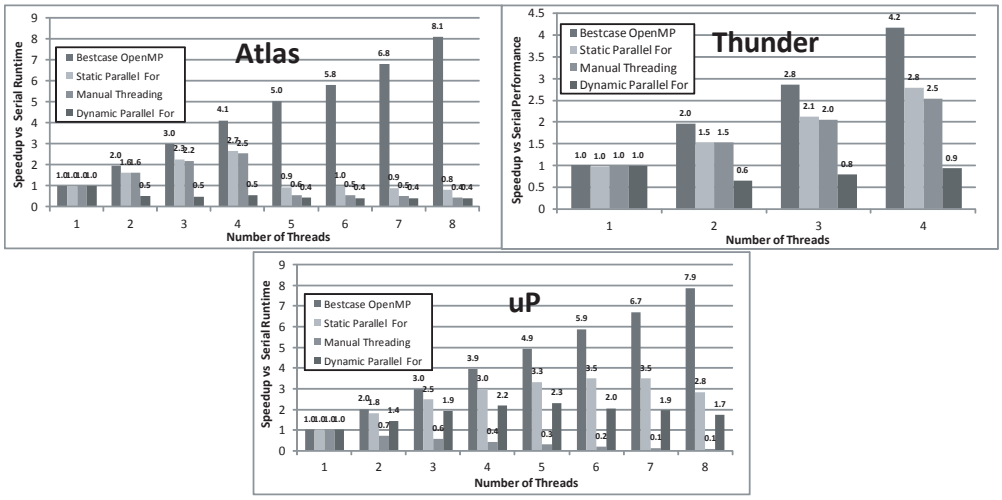

Fig. 6. CLOMP Untuned Default Scenario

The untuned results, shown in Figure 6, use the default run time environment variable settings, which is the most likely choice of application programmers. With these settings the (unrealistic) `best-case` configuration scales well up to 8 threads, which shows that good performance for the loop sizes common to multiphysics applications are possible. However, the realistic configurations all scale poorly, even causing increased run times in many cases.

The tuned results, shown in Figure 7, reflect the impact of changing environment settings so idle threads spin instead of sleep on uP and so idle threads spin much longer (`KMP_BLOCKTIME=100000`) before they sleep on Atlas and Thunder (we used these settings for the EPCC results presented in Section 4.1). These settings, which are appropriate for nodes dedicated to a single user, result in improved scaling for the `manual` and `for-static` scale configurations on both uP and Atlas. However, the actual speed ups, no more than 3.9, are still disappointing in light of the potential demonstrated by the `best-case` configuration. Further, the `for-dynamic` configuration still does not have sufficient work to compensate for the high overhead of the dynamic schedule kind. In fact, the "tuned" environment settings actually caused a slowdown for `for-dynamic` on uP and they did not improve performance on Thunder.

Figure 8 focuses on this effect by looking at the running times of several key components of CLOMP on uP when three major runtime parameters of IBM OpenMP are varied. Specifically, we show the cost in microseconds of `#pragma omp barrier` and cost of each of our four parallelization options in microseconds per loop iteration. The runtime parameters are:

- `Spins`: number of times each thread spin waits before it calls `yield`
- `Yields`: number of times a thread yields before it calls `sleep`
- `Delay`: time (unspecified units) between each scan of the work queue

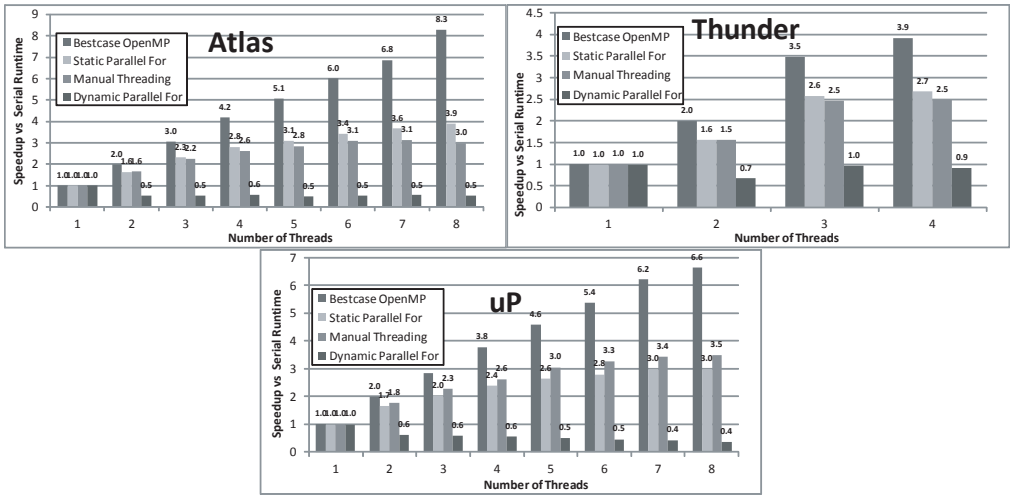

Fig. 7. CLOMP Tuned Default Scenario

For each parameter we evaluated three values: 1, 100 and infinity.

Figure 8 shows that the cost of barriers varies moderately with the different parameter values. Barriers are fastest when `Yields` is equal to infinity, regardless of the other parameters. `Spins` does not have a consistent effect on barrier performance. `Delay` is also inconsistent, although setting it to infinity leads to the worst barrier performance. The default configuration has average performance.

Looking at the parallelization variants, the parameter settings do not affect `best-case`, which uses no synchronization. The settings have a small impact on `manual` performance due to its barrier calls. In contrast, `for-static` and `for-dynamic` have much stronger dependence o nthe parameter settings. When `Yields` is equal to 1, `for-static` consistently performs poorly, while its performance improves dramatically with `Yields` set to 100 or infinity. The settings of `Delay` and `Spins` imply small differences in the performance of `for-static`. The cost of `for-dynamic` is much more unstable, with no parameter value consistently better than any other; the best performance occurs with `Yields` set to 100 and `Delay` to 1. Overall, different parameter settings provide the best performance with `for-dynamic` than with `for-static`, which complicates choosing appropriate defaults.

The Power5 processors on uP include SMT support that can run two hardware threads simultaneously, using hardware-level instruction scheduling. We evaluated the performance impact of using this feature by running CLOMP with 2 threads on each core. Figure 9 shows the relative speedup factor of the 2 threads per core configuration over 1 thread per core. Running 2 threads per core benefits `best-case` since it can utilize each core's functional units more fully. Overall, SMT can speed up `best-case` up to 60%. However, other configurations

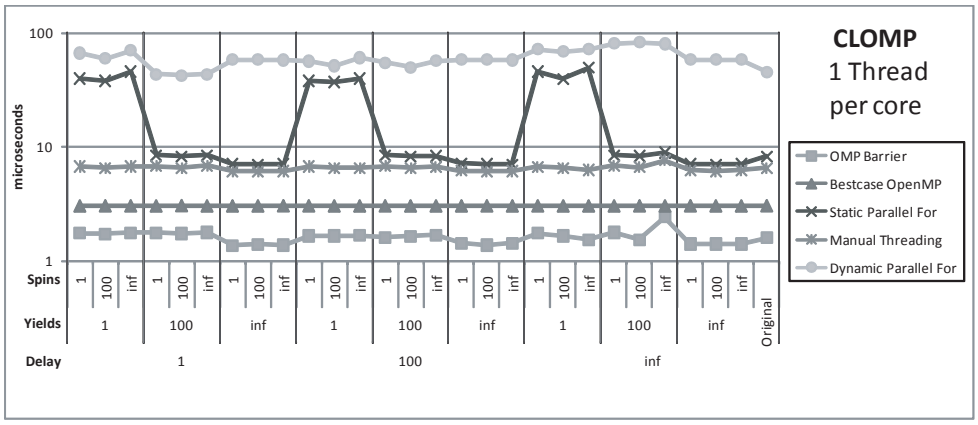

Fig. 8. CLOMP Performance on uP with full range of parameters, 1 thread per core

suffer in this mode. Barriers exhibit slowdowns between 20% and 30% because the interference between threads causes timing noise, which causes individual threads to arrive at barriers late, causing all threads to slow down. Similarly, `manual`, which combines `best-case` with barrier calls, runs slower since it runs faster between barriers but then must synchronize its threads at slower barriers. These performance drops are very small when `Yields` is set to infinity because this setting ensures that timing noise can never cause some thread to sleep while waiting at a barrier. However, the slowdown grows to more than 20% for other parameter settings. We observe similar behavior with `for-static` and `for-dynamic`, which perform best with `Yields` set to infinity, and consistently better with `Yields` set to 1 than 100 since the smaller setting reduces timing noise by keeping inactive threads from interfering with working threads. Finally, `for-dynamic` experiences less slowdown than `for-static` because its dynamic work allocation policy reduces load imbalance resulting from timing noise.

These results highlight the complexity of choosing the best OpenMP configuration, a task for which CLOMP results provide guidance. For our subsequent experiments we consistently used the modified OpenMP flags that optimized for the best performance of `for-static` rather than `for-dynamic` because the latter has much worse performance than the best of `manual` and `for-static`.

We examined the effects of memory bandwidth on the performance of parallel loops by increasing the number of zones per partition by a factor of 10 (1,000 zones per partition), which corresponds to some multiphysics application runs as well as some science codes. The results for this scenario, shown in Figure 10, exhibit outstanding scaling since the single core's memory bandwidth dominates performance of the sequential run. In fact, we observe superlinear speedups with `manual` and `for-static` on uP (e.g., 8.7x on with 8 threads) and on Atlas (peaking at 36x on 7 threads). The dramatic improvement on Atlas arises from the system's NUMA architecture, in which the penalty for accessing remote memory

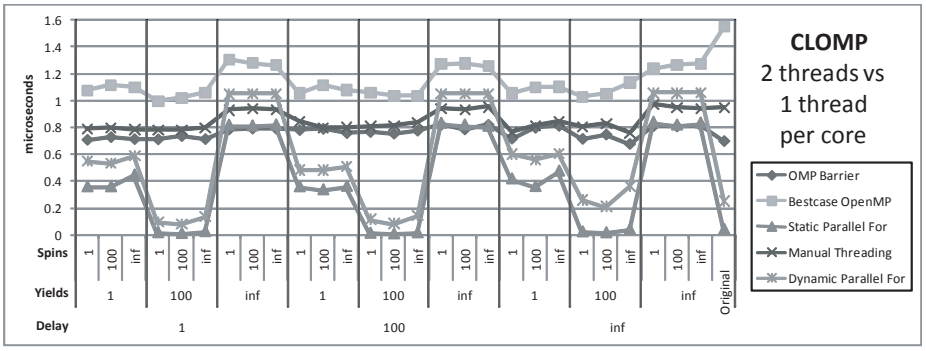

Fig. 9. CLOMP Relative speedup on uP of 2 threads per core vs 1 thread per core

via Hyper-Transport is relatively very high. Since the problem fits in cache with more threads, the performance benefit is significant. The cache effects are far smaller on uP and Thunder since these systems provide uniform memory access, with uP's slightly super-linear speedups attributable to its much larger cache. In all cases, these configurations are very close to the theoretical maximum of `best-case` while the `for-dynamic` configuration results continue to disappoint.

For application scenarios with even larger memory footprints, corresponding to science codes based on dense linear algebra routines, we no longer observe superlinear speedups since they no longer fit into cache. However, while we observe consistently good scaling on the uniform memory access systems, these scenarios provide insight into NUMA performance issues. Figure 11 shows results on Atlas for scenarios in which we increase the number of zones per partition over the default scenario 100x (10,000 zones per partition) and 1,000x (100,000 zones per partition). Here, we compare the two strategies for allocating application state: `serial`, where the master thread allocates all memory; and `threaded`, where each thread allocates its own memory. For each allocation strategy we show the speedup of the highest-performing realistic configuration. In both scenarios the two allocation strategies result in dramatically different performance, with the `threaded` allocation achieving near-linear speedup, while the `serial` allocation shows little improvement at all scales, similarly to previous observations on other NUMA systems. While application programmers generally will make the necessary coding changes to achieve these performance gains, the gains are not consistent: we still observed significant performance variation in our runs, with speed ups as low as 4 with eight threads. Examination of `/proc` data indicates that the threaded allocation does not guarantee the strict use of local memory. We are investigating using the `numactl` command in the NUMA library to provide more consistent performance.

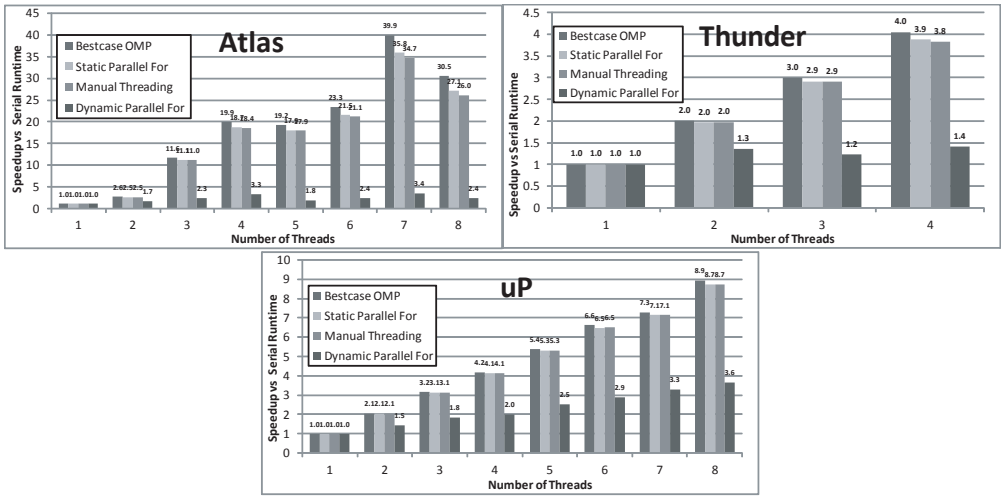

Fig. 10. CLOMP 10X Memory Scenario

By providing a `best-case` performance estimate, CLOMP puts the actual performance numbers in context of OpenMP overheads, cache effects, and NUMA effects. The `best-case` configuration is significantly different from the EPCC schedule test and represents a contribution of our work. For example, in Figure 10, the 27.1 speedup for 8 threads on Atlas is great but an even higher speedup of 30.5 was possible if the OpenMP overheads were lower. Similarly, the low `best-case` serial Allocation performance corresponding to the results in Figure 11 shows that OpenMP overhead is not the problem; NUMA effects are.

4.3 Studying the interactions of MPI and OpenMP

In this section, we evaluate the impact of MPI on Open MP overheads through the CLOMPI-No_Comm and CLOMPI-SendRecv variants of CLOMP using the same range of configurations on our experimental platforms. Since MPI implementations often use additional threads to monitor and to manage incoming and outgoing communication, the impact can be significant. On Thunder and Atlas the Quadrics and Infiniband network interface cards (NICs) provide additional processing power for these threads. Thus, their MPI implementations (Quadrics MPI and MVAPICH) can run these threads without interfering with the application, which is exactly the behavior we observed in our experiments on these platforms.

In constrast, uP's NICs do not support these threads, which requires IBM's MPI to run them on the same cores used for computation. However, the Power5 SMT capability allows MPI threads to share the cores with with CLOMP computational threads at little cost. We measured this effect by running CLOMPI-NoComm on uP, with 1 CLOMPI-NoComm thread per uP core. Figure 12 shows

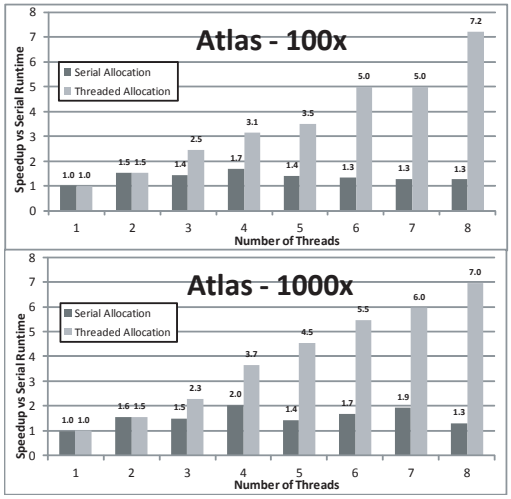

Fig. 11. CLOMP 100X and 1000X Memory Scenarios

the costs in microseconds of the key components of CLOMPI-NoComm (top) and the speedup of each component relative to CLOMP (bottom). Just running the additional MPI threads, as shown in CLOMPI-NoComm, leads `best-case` running 30% slower Since the MPI threads dfo not slow its barriers, `manual` fares a little better, slowing down only about 20%.

We observe a complex performance profile with `for-static`, which exhibits larger slowdowns with parameters that cause inactive threads to actively spin or yield rather than sleep (i.e., larger values of `Spins` and `Yields`). This behavior suggests that the MPI threads use functions such as `yield()` and `sleep()` to avoid using the CPU when other threads are active. Thus, the MPI threads can quickly determine that they have no work to perform when the main compute threads sleep soon after they have completed their work. In contrast, if the compute threads spin, the MPI threads awaken at random times and interfere with the compute threads.

Finally, the MPI threads cause little performance degradation with `for-dynamic` since dynamic scheduling adapts to non-deterministic interference. However, `for-dynamic` does incur a 20% slowdown when `Delay0` is set to infinity since the threads infrequently check the work queue, resulting in poor adaptation to interference

We evaluated the effect of MPI communication on OpenMP overheads by running CLOMPI-SendRecv, which adds an `MPI_Sendrecv()` call to `calc_deposit()`, on uP. Figure 13 shows the costs in microseconds of all CLOMPI-SendRecv components. While barriers have the same cost in CLOMPI-SendRecv and CLOMPI-NoComm, the various parallelization strategies cost more in absolute time with

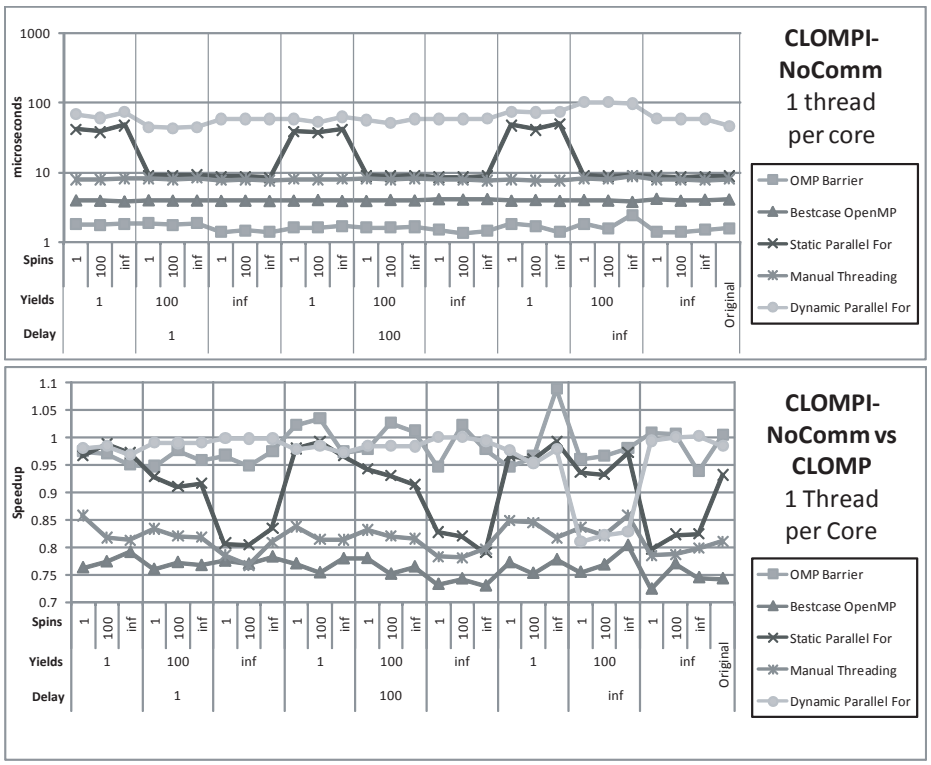

Fig. 12. CLOMPI-NoComm performance

CLOMPI-SendRecv since it does more work. However, it has essentially the same performance profile as CLOMPI-NoComm.

5 Conclusion and Future Work

Despite the popularity of shared memory systems and OpenMP's ease of use, overheads in OpenMP implementations and shared memory hardware have limited potential performance gains, thus discouraging the use of OpenMP. This paper presents CLOMP, a new OpenMP benchmark that models the behavior of scientific applications that have an overall sequential structure but contain many loops with independent iterations. CLOMP can be parameterized to represent a variety of applications, allowing application programmers to evaluate possible parallelization strategies with minimal effort and OpenMP implementors to identify overheads that can have the largest impact on real applications. Our results on three shared memory platforms demonstrate that CLOMP extends EPCC to capture the application scenarios necessary to characterize the impact of the overheads measured by EPCC. CLOMP guides selection of run time environment settings and can identify the impact of architectural features

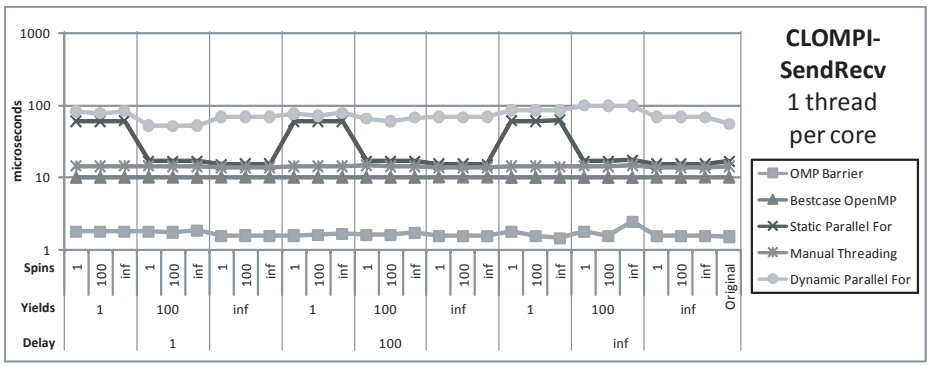

Fig. 13. CLOMPI-SendRecv performance

such as memory bandwidth, SMT and a NUMA architecture on application performance. The resulting insights can be very useful to application programmers in choosing the parallelization strategy and hardware that will provide the best performance for their application.

Overall, our results should not be seen as critiquing the OpenMP implementations that were used in our experiments. While we noted differences between them, the most significant issues arose from differences in the underlying architecture. Ultimately, CLOMP would provide its greatest value if it could guide architectural refinements that reduce the overheads of dispatching threads for OpenMP regions. For this reason, we included CLOMP in the benchmark suite of LLNL's Sequoia procurement.

References

1. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
2. L. Blackford, J. Choi, A. Cleary, E. Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammerling, G. Henry, A. Petite, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK Users. SIAM, Philadelphia, 1997.
3. V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *Proceedings of IEEE/ACM Supercomputing '04*, Nov. 2004.
4. W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. . Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The community climate system model version 3. *Journal of Climate*, 19(1):2122–2143, 2006.
5. J. D. de St. Germain, J. McCorquodale, S. Parker, and C. Johnson. A Component-based Architecture for Parallel Multi-Physics PDE Simulation. In *International Symposium on High Performance and Distributed Computing*, 2000.

6. R. Falgout, J. Jones, and U. Yang. *The Design and Implementation of HYPRE, a Library of Parallel High Performance Preconditioners*. Numerical Solution of Partial Differential Equations on Parallel Computers. Springer-Verlag, to appear.
7. T. Germann, K. Kadau, and P. Lomdahl. 25 Tflop/s Multibillion-Atom Molecular Dynamics Simulations and Visualization/Analysis on BlueGene/L. In *Proceedings of IEEE/ACM Supercomputing '05*, Nov. 2005.
8. F. Gygi, E. Draeger, B. de Supinski, R. Yates, F. Franchetti, S. Kral, J. Lorenz, C. Überhuber, J. Gunnels, and J. Sexton. Large-Scale First-Principles Molecular Dynamics simulations on the BlueGene/L Platform using the Qbox code. In *Proceedings of IEEE/ACM Supercomputing '05*, Nov. 2005.
9. J. Hoeflinger and B. R. de Supinski. The openmp memory model. In *International Workshop on OpenMP (IWOMP)*, 2005.
10. Message Passing Interface Forum. Mpi: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
11. OpenMP Architecture Review Board. OpenMP application program interface, version 2.5.
12. J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.
13. F. J. L. Reid and J. M. Bull. Openmp microbenchmarks version 2.0. In *European Workshop on OpenMP (EWOMP)*, 2004.
14. R. Rosner, A. Calder, J. Dursi, B. Fryxell, D. Q. Lamb, J. C. Niemeyer, K. Olson, P. Ricker, F. X. Timmes, J. W. Truran, H. Tufo, Y.-N. Young, M. Zingale, E. Lusk, and R. Stevens. Flash code: Studying astrophysical thermonuclear flashes. *Journal on Computing in Science and Engineering*, 2(2), 2000.
15. F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. de Supinski, J. Sexton, and J. Gunnels. 100+ TFlop Solidification Simulations on BlueGene/L. In *Proceedings of IEEE/ACM Supercomputing '05*, Nov. 2005.
16. B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the Computational Intensity of Unstructured Mesh Applications. In *Proceedings of the 19th ACM International Conference on Supercomputing*, June 2005.